A STUDY OF MEANING ALTERING MANIPULATION OF PROGRAMS

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the degree of
MASTER OF TECHNOLOGY

By
DILIP A. SONI

to the
COMPUTER SCIENCE PROGRAM
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

CERTIFICATE

CERTIFIED that the thesis entitled "A STUDY OF MEANING ALTERING MANIPULATION OF PROGRAMS" by Mr. Dilip A. Soni is a record of bonafide work carried out by him under my supervision and has not been submitted elsewhere for a degree.

H.V. Sahasrabuddhe

Kanpur
August 9, 1977

H.V. Sahasrabuddhe
Assistant Professor
Computer Science Program
and
Department of Electrical Engineering
Indian Institute of Technology
Kanpur (India)

ACKNOWLEDGEMENTS

I wish to express my deep gratitude to Dr. H.V. Sahasrabuddhe for generating and stimulating my interest in programming methodology, and for his constant understanding of my laziness.

I thank Mr. B.H. Jajoo for lively discussions and assistance during the course of the work; and Mr. R.B. Thakkar for drawing excellent diagrams.

Thanks are also due to Mr. H.K. Nathani for speedy and elegant typing and Mr. H.S. Tiwari for an excellent job of cyclostyling.

Finally acknowledgements are due to my dear friend A.K. Modi for no other reason but his wish that he be acknowledged.

Kanpur
August 8, 1977

- Dilip A. Soni

CONTENTS

	List of Figures	v
	Abstract	vi
Chapter 1	INTRODUCTION	1
Chapter 2	STEPWISE REFINEMENT AND OTHER TRANSFORMATIONS	3
Chapter 3	MEANING ALTERING TRANSFORMATIONS	8
Chapter 4	VARIATIONS	33
Chapter 5	CONCLUSIONS	51
	REFERENCES	54

LIST OF FIGURES

Figure No.	Figure	Page 2
2-1	Stepwise Refinement	5
2-2	Final Sort Program	6
3-1	Abstract Version of Solution for X	11
3-2	A Solution for X'	15
3-3	A Solution for X"	16
3-4	A Solution for X	18
3-5	Program after Transformation 3.1	22
3-6	Program After Transformation 3.2	24
3-7	Program after Transfermation 3.3	24
3-8	Abstract Program for Problem 3-2	26
3-9	A Solution for the Subprogram 3-2.1	29
3-10	A Solution for Subprogram 3-2.2	29
3-11	A Solution for Problem 3-2	31
4-1	Read a Card	34
4-2	Assemble and Print a Line	34
4-3	A Complete Solution for Printing from Cards	35
4-4	A Parallel Solution for Printing from Cards	36-37
4-5	Optimum Search Tree for the 31 Most Common English Words	39
4-6	Phase-1 of the Hu-Tucker Algorithm	42
4-7	Hardware Realization of Algorithm A	46
4-8	Hardware Realization of Algorithm B	48
4-9	Hardware Realization of Algorithm C	50

ABSTRACT

In the classical application of stepwise refinement for program development the successive refinements all attack the same problem. There are instances when related problems, which admit simpler solutions, are used to build up the solution of a complex programming problem. Such manipulations change the meaning of the intermediate programs and are named meaning altering transformations.

Use of meaning altering transformations has been illustrated through examples. In the process of program construction, both the program and its proofs undergo many cycles of parallel transformation before the final product is achieved. Variations in this techniques have been discussed.

While the utility of this method as a tool of explanation is evident, its efficiency as a programming technique could be questioned. A programming environment aiding systematic transformations may be needed, in order to practically program in this manner. Characteristics required of such an environment have been outlined. Integration of programs and parallel proof manipulation are some of its features.

1. INTRODUCTION

With the advance of modern electronic technology the power of computers has grown with a speed unmatched before. Along with this, the complexity and size of the programmer's task is also growing at the same rate. A programmer's task no longer consists in saving bits and microseconds but in being able to organize large and complex programs, assuring that they perform the function they are designed to perform. Inability to handle such complexity in programming reduces the confidence level of programs and a lot of effort and money is wasted on debugging.

With a view to master the complexity of programs and improve their reliability, a structure has been sought to be introduced in the programming activity. Structured programming has been used as a method of explaining programs as well as a programming technique. It enables a programmer to tackle problems systematically and confidently, and to produce better and more reliable programs.

In the classical application of the above method, successive refinements all attack the same problem. There exist several instances where related problems, which admit simpler solutions, are used to build up the solution of a complex programming problem. The proofs of correctness of the solutions of related problems, are used to derive the proof of correctness of the final solution. In the process of the program construction, both the program and its proof undergo many cycles of parallel transformations before the final product is achieved.

While the utility of this method as a tool of explanation is evident, its efficiency as a programming technique could be questioned. We need a programming environment aiding systematic transformations in order to practically program in this manner. The characteristics required of such an environment have been outlined.

Outline of the Thesis

Chapter 2 describes the method of program development with stepwise refinment, with an example. Chapter 3 introduces meaning altering transformations and explores their usefulness with the help of two examples. Related problems are solved first, and the solutions are combined to construct the solution for the original problem. Chapter 4 contains the variations of meaning altering transformations. Chapter 5 contains conclusions and outlines the areas in which future work may be necessary.

-

2. STEPWISE REFINEMENT AND OTHER TRANSFORMATIONS

Program development with stepwise refinement or structured programming has been one of the most recognized and talked about programming styles. Dijkstra[2] and Wirth[17,18,19] offer many examples of programs developed in this style. The style has been very lucidly described by Spier[14] in terms of construction of an imaginary picture from basic building blocks as follows:

Step 1: Act of design: Imagine the picture to be constructed vividly and completely.

Step 2: Modular decomposition: Further imagine the most convenient and natural lines along which the picture can be cut into a limited number of pieces.

Step 3: Modular decomposition by stepwise refinment. Treat each piece as if it were the original picture and repeatedly apply steps 1,2 and 3 until you deal with a manageable number of elementary pieces, which can be trivially constructed with basic building blocks.

Step 4: Bottom up construction: Construct the elementary pieces with basic building blocks. Use these in turn as building blocks for next level of pieces. Repeat step 4 until the entire picture has been assembled.

Let the program to be constructed be the imaginary picture, the elementary pieces be the primitive subroutines and the basic building blocks be program statements. Call the first three steps 'top down design', and the last step 'bottom up implementation' and we have the method called structured programming.

We shall take up an example to illustrate the programming style just described. This example and the style of solution have been borrowed from Wirth[19].

Example 2-1: Sequential Merging

A set of $n = 2^b$ integer variables is to be sorted using an array, A, of size 2n. The variables are initially stored in A[1] through A[n], and may be viewed as sequences of length one in correct order. The variables are shuttled between A[1] through A[n] and A[n+1] through A[2n], merging pairs of sequences in ascending order during each move.

Let the variable 'P' denote the size of merged sequences. Let i,j and k,l denote the pairsof access points for input part of A and output part of A respectively.

Let 'up' denote the direction of movement of variables. When 'up' is true, variables are transferred from A[1] through A[n] to A[n+1] through A[2n], and vice-versa when 'up' is false.

Repeated application of steps 1, 2 and 3 appear in Figure 2-1.

The pieces in Figure 2-1 are assembled to construct the final version of the sort program, as it appears in Figure 2-2. Though suitable for compilation, it is no longer as understandable as the scattered pieces in Figure 2-1 are. If, however, the development process as depicted in Figure 2-1 is preserved for documentation, one can easily follow the transformations carried out on the top level abstract program to understand the final program in Figure 2-2.

All transformations, used in the above example, keep the meaning of the program unaltered. We use the term 'refinment' for the kind of transformations used in the example.

```
up := true; p := 1;
repeat
   1: "initialize indices i,j,k, and l";
   2: "merge p-tuples from i-and j-sequences into k-and l-sequences"
      up := ¬ up;  p := 2*p;
until p = n
```

(a) Top level program

```
1: if up then i := 1; j := n;
             k := n+1; l := 2*n;
          else k := 1; l := n;
          e    i := n+1, j := 2*n
```

(b) Piece 1 : Initialize

```
2: m := n, h := 1;
   repeat
      m := m - 2 * p;
      2.1: "merge one p-tuple from each of i and j to k,
            increment k after each move by h";
      2.2: "exchange k and l"
   until m = 0
```

(c) Piece-2: Merge p-tuples.

```
2.1: q := p; r := p;
     repeat   select the smaller item
        if A[i ] < A j   then A[k]  := A[i ];
                              k := k+h; i := i+1; q := q-1;
                         else A[k ] := A[j ];
                              k := k+h; j := j-1; r := r-1:
        fi
     until (q=0) V (r=0)
     2.1.1:"copy tail of i-sequence";
     2.1.2:"copy tail of j-sequence";
```

(d) Piece 2.1: Copy one pair of p-tuples

```
2.2 t := k; k := l; l := t;
```

(e) piece : 2.2: Exchange k and l

```
2.1.1: while q ≠ 0 do
       begin
          A[k ] := A[i ];
          k := k+h; i := i+1; q :=q-1;
       end
```

(f) piece 2.1.1: Copy tail of i sequence.

```
2.1.2: while r ≠ 0 do
       begin
         A[k ]:= A[ j ];
         k := k+h; j := j-1; r := r-1;
       end
```

```
up := true; p := 1;
repeat
   if up then i := 1; j := n;
              k := n+1; l := 2*n;
         else k := 1; l := n;
              i := n+1; j := 2*n;
   fi
   m := n, h := 1;
   repeat
      m := m - 2 * p;
      q := p; r := p;
      repeat
         if A[i ]<A[j ] then A[k ] := A[ i];
                             k := k+h; i := i+1; q := q - 1;
                        else A[k ]: = A[j ];
                             k := k+h; j := j-1; r := r - 1;
         fi
      until (q=0) V (r=0).
      while q ≠ 0 do
      begin
         A[k ] := A [i ];
         k := k+h; i := i+1; q := q-1;
      end
      while r ≠ 0 do
      begin
         A[k ] := A[j ];
         k := k+h; j := j-1; r := r-1;
      end
   until m = 0
   up := ¬up; p := 2*p;
until p = n
```

Figure 2-2: Final Sort Program.

There is another class of transformations, which also preserve program equivalence. They aid in optimizing the programs to meet certain operational constraints. Many optimizing compilers carry out such transformations to optimize programs written in a higher level programming language. Schaefer[10] offers a theory of such transformations while the I.B.M. Program Transformation Catalogue[15] is a source book of source-to-source transformations.

3. MEANING ALTERING TRANSFORMATIONS

Another class of transformations are manually applied to programs in computer installations all over the world. Old programs are updated or transformed to perform additional functions. These transformations do change the meaning of the programs. Often, simple programs become more and more complex as a series of transformations are applied to them over a period of time.

The following is a parallel in rocket engineering quoted by Knuth(9). If one takes a cross section of the German V-2 rocket of World War II fame, one finds external skin, structural rods, tank wall etc. On the other hand, if one cuts across the Saturn-B moon rocket, one finds only an external skin, which also serves as a structural component and the tank wall as well. Several functions have been integrated in one part of the rocket.

The V-2 rocket would never have been airborne if its designers had originally tried to combine all its function. Only after studying the simple V-2 rocket, were they in a position to transform it into a sophisticated Saturn-B rocket. Perhaps, they required a series of transformations to arrive at the final result.

We can find many examples in all disciplines, where simple systems are transformed into more complex and sophisticated systems. Sometimes, a number of systems are also combined to yield a bigger and more versatile system.

Though used in program maintenance, such meaning altering transformations have rarely been used in program development. One example in literature is the garbage collection algorithms of Knuth[7]. Two conceptually simple but inefficient algorithms are integrated to give a difficult to understand, but efficient algorithm. Sahasrabuddhe[12] states that, 'If transformations are to become a programming style, the class of available transformations would have to be expanded to include some which do alter meaning'.

To explore the use of meaning altering transformations in program development, several examples have been worked out using meaning preservin as well as meaning altering transformations.

The programming language in the following examples, is a pseudo Pascal[18] like language. Proof technique of Hoare[3] has been used to prove the correctness of the programs.

Example 3-1

A set X is defined as follows:

(i) $1 \in X$

(ii) If $y \in X$ then $2 * y + 1 \in X$

and $3 * y + 1 \in x$ (3-1)

(iii) No other element belongs to X.

Generate first N elements of X in ascending order.

We shall assume that the elements of X are to be stored in an array named SETX.

The input-output conditions to be satisfied by a program to solve the above problem appear below.

Initial condition : true (3-2)

Final condition : SETX[1] ... SETX[N] are first N elements of set X, in ascending order. (3-3)

We can divide the final condition into several parts:

(i) SETX[1] .... SETX[N] are elements of X, that is

$$R_1 \equiv (\forall j : 1 \leq j \leq N)\ (SETX[j] \in X) \tag{3-4}$$

(ii) SETX[1] ... SETX[N] are in ascending order, that is

$$R_2 \equiv (\forall j : 1 \leq j < N)(SETX[j] < SETX[j+1]) \tag{3-5}$$

(iii) SETX[1] ... SETX[N] are first N elements of X, that is

$$R_3 \equiv \neg((\exists k)(k \in X) \wedge (k < SETX[N]) \wedge ((\forall m : 1 \leq m \leq N)(k \neq SETX[m]))) \tag{3-6}$$

This R_3 is the most complex condition of the three and it is not obvious how we may establish it in a program.

The final condition may now be re-written as follows:

$$R \equiv R_1 \wedge R_2 \wedge R_3 \tag{3-7}$$

The top level abstract version of the program appears in Figure 3-1 where P is an invariant yet to be derived.

We can generalize the final condition R to obtain P. P holds where i N elements have been stored in SETX.

$$P \equiv P_0 \wedge P_1 \wedge P_2 \wedge P_3 \tag{3-8}$$

where

$$P_0 \equiv (1 \leq i \leq N) \tag{3-9}$$

$$P_1 \equiv (\forall j : 1 \leq j \leq i)(SETX[j] \in X) \tag{3-10}$$

$$P_2 \equiv (\forall j : 1 \leq j < i)(SETX[j] < SETX[j+1]) \tag{3-11}$$

$$P_3 \equiv \neg((\exists k)(k \in X) \wedge (k < SETX[i]) \wedge ((\forall m : 1 \leq m \leq i)(k \neq SETX[m]))) \tag{3-12}$$

It is easy to establish that

```
true
'initialize'
{ P }
while i ≠ N do
begin
    'a step towards i = N under invariance of P'
end
{ P ∧ (i = N) }
{ R }
```

Figure 3-1: Abstract version of solution for X.

At this point, we make a departure from the conventional stepwise refinment methodology. Two simpler versions of the problem are introduced and solved next. Following that, a solution to the original problem will be derived from the solutions to the simpler problems.

Simplified Problems

Structurally the simpler problems are quite similar to the original. Two sets X' and X" are defined. The modified sets are generated in the simplified problems using the same approach as above. What makes the problem simpler, then, is the definition of the new sets.

The set X' is defined as follows:

(i) $1 \in X'$

(ii) If $y \in X'$ then $2 * y + 1 \in X'$ (3-14)

(iii) No other element belongs to X'

The modified problem is to generate the first N elements of X', in ascending order.

The final condition and the invariant may now be rewritten as follows:

$$R^1 \equiv R[X \rightarrow X'] \quad (3\text{-}15)$$

$$P^1 \equiv P[X \rightarrow X'] \quad (3\text{-}16)$$

Let us now study P_3^1 closely. In full, P_3^1 looks as follows:

$$P_3^1 \equiv \neg((\exists k)(k \in X') \wedge (k < SETX[i]) \wedge ((\forall n : 1 \leq n \leq i)(k \neq SETX[n]))) \quad (3\text{-}17)$$

It states that if there is an element of X', which is smaller than SETX[i], it must be one of the elements stored in SETX. From the definition of X' it is clear that if an element of X' is smaller than SETX[i], it can be directly generated by one of the elements in SETX.

In other words,

$$(k \in X') \wedge (k < \text{SETX}[i]) \Rightarrow (\exists n)(1 \leq n < i) \wedge (k = 2 * \text{SETX}[n] + 1) \tag{3-18}$$

At this stage, we ask which element of X' can be the next element to enter SETX. Answer is that it must be greater than SETX i and can be directly generated by one of the elements in SETX.

If we also store the elements, which are directly generated by elements in SETX and are greater than SETX i , in ascending order, we achieve the following:

(i) It is now easy to establish P_3^1

(ii) Next element to enter SETX is the minimum of above elements.

Since both SETX and the newly proposed storage are generated in ascending order, a first in first out (FIFO) queue is the natural data structure for the new storage.

With the help of these ideas, P^1 may be rewritten as follows:

$$P^2 \equiv P_1^2 \wedge P_2^2 \wedge P_3^2 \wedge P_4^2 \wedge P_5^2 \tag{3-19}$$

where

$$P_1^2 \equiv P_1^1 \tag{3-20}$$

$$P_2^2 \equiv P_2^1 \tag{3-21}$$

$$P_3^2 \equiv P_3^1 \tag{3-22}$$

$$P_3^2 \equiv ((\forall j : 1 \leq j \leq i) \wedge (y = 2*\text{SETX}[j] + 1)) \tag{3-23}$$

$$(((\exists n)(1 \leq n \leq i) \wedge (y = \text{SETX}[n]))$$

$$\vee((y > \text{SETX } i) \wedge (y \in Q2))$$

$P_4^2 \equiv$ all elements of Q2 are in ascending order (3-24)

$P_5^2 \equiv$ No other element belongs to Q2 (3-25)

Q2 is a FIFO queue.

After the above analysis, it is easy to prove that the program in Figure 3-2 correctly solves the problem of generating first N elements of X'.

Similarly the set X" is defined as follows:

(i) $1 \in X''$

(ii) If $y \in X''$ then $3 * y + 1 \in X''$ (3-26)

(iii) No other element belongs to X".

The problem is to generate the first N elements of X", in ascending order.

We can get the final condition R^3 and invariant P^3 after making the following substitutions in R^1 and P^2 respectively.

$$R^3 \equiv R^1 \; [\, X' \longrightarrow X'' \,] \tag{3-27}$$

$$P^3 \equiv P^2 \; [\, X' \longrightarrow X'', \; Q2 \longrightarrow Q3, \; (y = 2 * \text{SETX}[j] + 1) \longrightarrow (x = 3 * \text{SETX}[j] + 1) \,] \tag{3-23}$$

A program for constructing the beginning of X" can be written on the same line as before. Such a program is given without further comments as Figure 3-3.

```
{true}
i, SETX [1]  := 1,1;
Q2 ⇐ 2 * SETX[i] + 1;
{P^2 has been established}
while i ≠ N do
begin
  i := i + 1;
  SETX [i] := head (Q2); pop (Q2);
  Q2 ⇐ 2 * SETX [i] + 1;
end
{P^2 ∧ (i = N)}
{R^1}
```

Figure 3-2: A solution for X'

---

where

head (Q2) is the element at the head of the queue Q2.

Q2 y ≡ an element with value y joins Q2.

pop (Q2) ≡ element at the head of Q2 is removed.

```
{true}
 i, SETX[ 1 ] := 1,1;
 Ø ⇐ 3*SETX[i] +1;
{P^3 has been established}
 while i ≠ N do
 begin
    i := i + 1;
    SETX [i ] := head (Ø); pop (Ø);
    Ø ⇐ 3 * SETX [ i ]+ 1;
 end
{P^3 ∧ (i=N)}
{R^3}
```

Figure 3-3: A Solution for X''

Solution of the Original Problem

We now ask ourselves if we can simply merge the programs in Figures 3-2 and 3-3 to obtain one for X. In fact this is possible except for the place where the next element to enter SETX is picked. At that point, we do not know whether to write

SETX[i] := head(Q2);

or SETX[i] := head (Q3);

or both.

We want to add the smallest eligible element to SETX. Thus, if head (Qa) is smaller than head (Qb) then the right action is

SETX [i] := head (Qa); POP(Qa);

If, however, head (Q2) is equal to head (Q3), then we must include just one copy of this element in SETX, and delete it from both queues:

SETX[i] := head (Q2); POP(Q2); POP(Q3);

with a little exercise, we can write the invariant for the new program:

$$P^4 \equiv P_0 \wedge P_1 \wedge P_2 \wedge P_3^4 \wedge P_4^4 \wedge P_5^4 \tag{3-29}$$

where P_0, P_1 and P_2 are as defined in (3-9), (3-10), and (3-11).

$$P_3^4 \equiv P_3^2 \wedge P_3^3 \tag{3-30}$$

$$P_4^4 \equiv P_4^2 \wedge P_4^3 \tag{3-31}$$

$$P_5^4 \equiv P_5^2 \wedge P_5^3 \tag{3-32}$$

We claim that the program in Figure 3-4 correctly solves the original problem.

```
{true}
i, SETX [1] := 1,1;
Q2 ⇐ 2 * SETX [i] + 1;   Q3 ⇐ 3*SETX [i] + 1;
{P4}
while i ≠ N do
begin
   i := i + 1;
  {S1}
   if head (Q2) < head (Q3) then SETX [i] := head (Q2), pop(Q2);
   else if head (Q2) = head (Q3) then SETX[i] := head (Q2);
                                      pop(Q2); pop(Q3);
   else if head (Q2) > head (Q3) then SETX[i] := head (Q3);
                                      pop(Q3);
   fi
  {S2}
   Q2 ⇐ 2 * SETX [i] + 1; Q3 ⇐ 3 * SETX [i] + 1;
  {S3}
end
{P4 ∧ (i = N)}
{R}
```

Figure 3-4: A solution for X.

Proof of Correctness

We shall now prove the correctness of the program in Figure 3-4.

1. Establishment of P^4 before the while loop is trivial.

2. We have to prove that the execution of the statement inside the while loop leaves P^4 invariant.

We shall first derive the assertions S_1, S_2 and S_3, as positioned in Figure 3-4.

(i) S_1 :

$$\{P^4 \wedge (i \neq N)\}$$
$$i := i + 1; \qquad (3\text{-}33)$$
$$\{S_1 \equiv P_0 \wedge P^4 [i \rightarrow i - 1]\}$$

(ii) S_2 : We have to consider three cases.

Case 1: head (Q_2) $\prec$ head (Q_3)

Here, head (Q_2) is the minimum of the two heads and hence, it is the next element to enter SETX. It should be removed from Q_2.

$$\{S_1\}$$
$$SETX[i] := head\ (Q_2);$$
$$POP(Q_2); \qquad (3\text{-}34)$$
$$\{S_1 \wedge P_1 \wedge P_2 \wedge P_5^4\}$$

Case 2: head (Q_2) = head (Q_3)

Here both heads are equal and only one of them should be entered in SETX. However, both of them are removed from respective queues.

$$\{S_1\}$$
$$SETX[i] := head\ (Q_2);$$
$$POP\ (Q_2);\ POP\ (Q_3); \qquad (3\text{-}35)$$
$$\{S_1 \wedge P_1 \wedge P_2 \wedge P_5^4\}$$

Case 3: head ($\mathcal{Q}2$) $>$ head ($\mathcal{Q}3$)

Argument for this case proceeds along the lines of Case 1 above.

Combining the three cases, we conclude

$$\{S_1\}$$
$$\underline{\text{if}}$$
$$\vdots$$
$$\underline{\text{fi}}$$
$$\{S_2 \equiv S_1 \wedge P_1 \wedge P_2 \wedge P_5^4\} \tag{3-36}$$

(iii) Two new elements 2 * SETX[i] + 1 and 3 * SETX[i] + 1 are generated by SETX[i] . and are entered in $\mathcal{Q}2$ and $\mathcal{Q}3$. Since they are greater than 2 * SETX[i - 1] and 3 * SETX[i - 1] respectively, $\mathcal{Q}2$ and $\mathcal{Q}3$ are still in ascending order.

$$\{S_2\}$$
$$\mathcal{Q}_2 \Leftarrow 2 * \text{SETX}[i] + 1; \quad \mathcal{Q}3 \Leftarrow 3 * \text{SETX}[i] + 1;$$
$$\{S_3 \equiv S_2 \wedge P_3^4 \wedge P_4^4\} \tag{3-37}$$
$$\{P_3^4\}$$

Therefore, execution of the <u>while</u> loop leaves P invariant.

3. We have to prove now that the program terminates. The value of 'i' increases by one each time the <u>while</u> loop is executed and i equals N after N - 1 executions. Therefore, we conclude that the program terminates.

From the above arguments, we have succeeded in showing that the program in Figure 3-4 is correct.

Optimization

We shall now apply several transformations to our program, in order to get a lower level implementation and to make it efficient.

Transformation 3.1.5

Instead of storing generated elements in the queues $Q2$, $Q3$, we shall store generating elements

The invariant P^4 may be transformed as follows:

$$P^5 \equiv P^4[\ y \in Q2 \rightarrow SETX[j] \in Q2,\ z \in Q3 \rightarrow SETX[j] \in Q3] \quad (3\text{-}38)$$

We make the following substitutions to obtain the program in Figure 3-5.

$$\begin{aligned}
&\text{(i) head } (Q2) \longrightarrow 2 * \text{head } (Q2) + 1 \\
&\text{(ii) head } (Q3) \longrightarrow 3 * \text{head } (Q3) + 1 \qquad (3\text{-}39) \\
&\text{(iii) } Q2 \Leftarrow 2 * SETX[i] + 1; \longrightarrow Q2 \Leftarrow SETX[i]; \\
&\text{(iv) } Q3 \Leftarrow 3 * SETX[i] + 1; \longrightarrow Q3 \Leftarrow SETX[i];
\end{aligned}$$

Transformation 3.2

With $Q2$ and $Q3$ as defined after transformation 1, we observe that they are final subsequences of SETX. Therefore, they may be implemented on the array SETX with the help of pointers to the heads of the queues.

The invariant P^5 may be transformed as follows:

$$P^6 \equiv P_0 \wedge P_1 \wedge P_2 \wedge P_3^6 \quad (3\text{-}40)$$

where

$$P_3^6 \equiv P_3^4[\ y \in Q2 \rightarrow j \geq Q2PTR,\ z \in Q3 \rightarrow j \geq Q3PTR\ ] \quad (3\text{-}41)$$

$Q2PTR$ and $Q3PTR$ are pointers to the heads of $Q2$, $Q3$ respectively.

```
{true}
1, SETX[1] := 1,1;
Q2 ⇐ SETX[i]; Q3 ⇐ SETX[i];
{P5}
while i ≠ N do
begin
   i := i + 1;
   if      2 * head(Q2)+1 < 3*head(Q3)+1 then SETX[i] := 2*head(Q2)+1;
                                              pop(Q2);
   else if 2 *head(Q2)+1 = 3*head(Q3)+1 then SETX[i] := 2*head(Q2)+1;
                                              pop(Q2); pop(Q3);
   else if 2 *head(Q2)+1 > 3*head(Q3)+1 then SETX[i] := 3*head(Q3)+1;
                                              pop(Q3);
   fi
   Q2 ⇐ SETX[i] ; Q3 ⇐ SETX[i] ;
end
{P5 ∧ (i = N)}
{R}
```

Figure 3.5: Program after Transformation 3.1

We make the following substitutions in Figure 3-5.

(i) head Q2 ⟶ SETX Q2PTR

(ii) head Q3 ⟶ SETX Q3PTR

(iii) POP(Q2); ⟶ Q2PTR := Q2PTR + 1; (3-42)

(iv) POP(Q3); ⟶ Q3PTR := Q3PTR + 1;

(v) Q2 ⟸ SETX[i] ; ⟶ empty

(vi) Q3 ⟸ SETX[i] ; ⟶ empty

Besides the above substitutions, the initialization

Q2PTR, Q3PTR := 1,1;

is needed to establish P^6 initially. As a result, the program in Figure 3-6 is obtained.

Transformation 3.3

In Figure 3-6 we see that the expression 2 * SETX [Q2PTR]+ 1 and 3 * SETX [Q3PTR] + 1 are frequently computed. These frequent computations can be avoided through the standard technique of common subexpression elimination[13,16]. The result of such elimination appears in Figure 3-7.

The beginning steps of our development of the program were on the lines of classical stepwise refinement. Later, however, a deviation occurred when we solved two simpler problems first, which made the 'discovery' of the appropriate higher level information structure natural.

Another point to note is the introduction and subsequent elimination of the queues. The queues here are an explanatory device, which luckily has not affected the efficiency of the final program.

```
{true}
 i, SETX[1]  := 1,1;
 Q2PTR, Q3PTR := 1,1;
{P^6}
 while  i ≠ N do
 begin
    i : = i+1;
    if 2*SETX Q2PTR  +1 < 3*SETX[Q3PTR] +1 then SETX[i ] :=  2*SETX [Q2PTR]+1
                                                Q2PTR := Q2PTR+1;
    else if 2*SETX [Q2PTR]+1 = 3*SETX [Q3PTR ]+1 then SETX i  := 2*SETX[ Q3PTR+
                                                    ,PTR)+1 ;
                                                Q2PTR := Q2PTR+1; Q3PTR :=
                                                    Q3PTR:=Q3PTR+1;
    else if 2*SETX[ Q2PTR]+1 >3*SETX[ Q3PTR]+1  then SETX[i ]:= 3*SETX[ Q3PTR]
                                                Q3PTR := Q3PTR+1;
    fi
 end
{P^6 ∧ (i=N)}
{R}
```

Figure 3-6: Program after transformation 3.2

```
{true}
 i, SETX[1 ]  := 1,1;
 Q2PTR; Q3PTR := 1,1;
 X2 := 2*SETX [ Q2PTR] +1; X3  := 3*SETX[ Q3PTR ]+ 1;
{P^6}
 while i ≠ N do
    i := i+1;
    if X2 < X3  then  SETX[i ]  := X2;
                      Q2PTR := Q2PTR+1; X2  := 2* SETX[ Q2PTR ] + 1;
    else if X2=X3 then SETX[i ]  := X2;
                      Q2PTR := Q2PTR+1; X2  := 2*SETX[ Q2PTR]+1;
                      Q3PTR := Q3PTR+1; X3 := 3*SETX[ Q3PTR] + 1;
    else if X2 > X3 then SETX[i ]  := X3;
                         Q3PTR := Q3PTR+1; X3  := 3*SETX [ Q3PTR] + 1;
    fi
 end
{P^6 ∧ (i=N)}
{R}
```

Figure 3-7: Program after transformation 3.3

Later we derived Figure 3-4 from Figures 3-2 and 3-3, using a transformation, which doesnot preserve meaning. In fact, its purpose is to alter the objective sets of the programs involved. While the transformation involved in deriving Figure 3-5 from Figure 3-4, does not alter the meaning of the program as a whole, it does change the meaning of the information structure. This change in the meaning the information structure enables application of further optimizing transformations to Figure 3-5. These optimizing transformations could in theory be performed mechanically.

In the process of construction outlined, proof of correctness of programs also undergoes parallel transformations, along with programs, to yield the final proof of correctness. This is an interesting development and may be useful in verification of programs.

The observations made above are further illustrated by the following example.

Example 3-2

Sort a set of N integer variables in ascending order. All the variables are either 1's, 2's or 3's. The variables are initially stored in an array A, of size N.

The initial and final conditions to be satisfied by the program appear below:

Initial condition

$$I \equiv (\forall j : 1 \leq j \leq N)(A[j] = 1) \vee (A[j] = 2) \vee (A[j] = 3) \qquad (3\text{-}43)$$

final condition

$$\begin{aligned} R \equiv (\exists \text{ head2 and head3})(&1 \leq \text{head2} \leq N+1) \wedge (1 \leq \text{head3} \leq N+1) \\ &((\forall j : 1 \leq j < \text{head2})(A[j] = 1)) \\ &((\forall l : \text{head2} \leq l < \text{head3})(A[l] = 2)) \\ &((\forall k : \text{head3} \leq k \leq N)(A[k] = 3)) \end{aligned} \qquad (3\text{-}44)$$

R states that there exist head22 and head33, two points of accession points, which partition the array A into three parts containing 1's, 2's and 3's respectively.

An abstract version of the program appears in Figure 3-8, where P is an invariant yet, to be derived.

```
{I}
'Initialize';
{P}
while k ≠ N do
begin
    'a step towards k = N under invariance of P';
end
{P ∧ (k = N)}
{R}
```

Figure 3-8: Abstract program for Problem 3-2

As in the previous example, we shall first solve two simplified problems.

Simplified Problem 3-2.1

Sort a set of integer variables in ascending order. All the variables are either 1's or 2's.

The initial and final conditions to be satisfied by the program appear as below.

Initial condition

$$I^1 \equiv (\forall j : 1 \leq j \leq N)(A[j] = 1) \vee (A[j] = 2) \qquad (3\text{-}45)$$

Final condition

$$R^1 \equiv (\exists \text{head22})(1 \leq \text{head22} \leq N+1)$$
$$((\forall j : 1 \leq j < \text{head22})(A[j] = 1)) \qquad (3\text{-}46)$$
$$((\forall l : \text{head22} \leq l \leq N)(A[l] = 2))$$

Interpretation of R^1 is similar to that of R. Abstract version of the program may be obtained by replacing the assertions I, R and R^1 in Figure 3-8 with I^1, P^1 and R^1 respectively.

We may transfer all 1's to the lower part of the array A to sort the given elements.

The invariant P^1 may be written as a generalization of R^1 as follows:

$$P^1 \equiv (1 \leq i \leq N+1) \wedge (0 \leq k \leq N) \wedge (k = i - 1)$$
$$((\exists \text{head2}) \quad (1 \leq \text{head2} \leq i) \tag{3-47}$$
$$\wedge ((\forall j : 1 \leq j < \text{head2})(A[j] = 1))$$
$$(\forall l : \text{head2} \leq l < i)(A[l] = 2)))$$

With the above invariant, it is easy to prove the correctness of the program in Figure 3-9.

Simplified Problem 3-2.2

Sort a set of integer vairiables in ascending order. All the variables are either 2's or 3's.

Preceeding as before, the initial and final conditions may be written as follows:

Initial condition

$$I^2 \equiv (\forall j : 1 \leq j \leq N)(A[j] = 2) \vee (A[j] = 3) \tag{3-48}$$

Final condition

$$R^2 \equiv (\exists \text{head3})(1 \leq \text{head3} \leq N + 1)$$
$$\wedge ((\forall l : 1 \leq l < \text{head3})(A[l] = 2)) \tag{3-49}$$
$$\wedge ((\forall m : \text{head3} \leq m \leq N)(A[m] = 3))$$

Abstract version of the program may be obtained by replacing the assertions I, P and R in Figure 3-8 with I^2, P^2 and R^2 respectively.

We shall sort the elements by transferring all 3's to upper part of the array A.

The invariant P^2 may be written as follows:

$$\begin{aligned} P^2 \equiv\ & (1 \leq i \leq N+1) \wedge (0 \leq k \leq N) \\ & ((\exists\ \text{head3})(i \leq \text{head3} \leq N+1) \\ & ((\forall l : 1 \leq l < i)(A[l] = 2)) \\ & ((\forall n : \text{head3} \leq j \leq N)(A[j] = 3))) \end{aligned} \tag{3-50}$$

Using the above algorithms and the invariant, we may write the required program as in Figure 3-10.

To prove the correctness of the program, it is sufficient to note that

$$k = i + (N - \text{head3}) \tag{3-51}$$

This trivially follows from the code in the while loop in Figure 3-10, where in each execution of the loop, either 'i' is incremented or 'head3' is decremented, by one.

Also,

$$P^2 \wedge (k = N) \wedge (k = i + (N - \text{head3})) \Longrightarrow R^2 \tag{3-52}$$

With the above analysis, it is to prove the correctness of the program in Figure 3-10.

Solution of Original Problem 3-2

We observe that

$$R^1 \wedge R^2 \Longrightarrow R \tag{3-53}$$

$$P \equiv P^1 \wedge P^2 \tag{3-54}$$

```
{I^1}
 k,i, head2 := 0,1,1;
{P^1}
 while k ≠ N do
 begin
    if A[i] = 1 then A[i], A[head2] := A[head2], A[i];
                     head2 := head2+1; i := i+1;
    else if A[i] = 2 then i := i + 1;
    fi
    k := k + 1;
 end
{P^1 ∧ (k = N)}
{R^1}
```

Figure 3-9: A solution for the subprogram 3-2.1

```
{I^2}
 k,i, head3 := 0,1,N + 1;
{P^2}
 while k ≠ N do
 begin
    if A[i] = 2 then i := i + 1;
    else if A[i] = 3 then head3 := head3 - 1;
                          A[i], A[head3] := A[head3], A[i];
    k := k+1;
 end
{P^2 ∧ (k=N)}
{R^2}
```

Figure 3-10: A solution for subprogram 3-2.2.

and simply merge the two solutions, we get the required solution to the original problem and it appears in Figure 3-11.

The proof of correctness of the above solution is quite easy and is left to the reader.

Optimizing Transformation 3-2.1

From (3-51)

$$(k = N) \wedge (k = i + (N - head3)) \Rightarrow (i = head3) \qquad (3\text{-}55)$$

Since the variable 'k' is not used actively any where in the program, it may be eliminated and the loop terminating condition $(k = N)$ may be replaced by $(i = head3)$.

Transforming the invariant P, we get

$$P^3 \equiv P\,[(0 \leq k \leq N) \longrightarrow \underline{true},\ (k = i - 1) \longrightarrow \underline{true}\,] \qquad (3\text{-}56)$$

Applying this transformation to Figure 3-11, we get Figure 3-12.

As in the solution of the Problem 3-1, we obtain two problems by limiting the values of the variables. Solutions to the above problems, along with their proofs of correctness were then combined to build the solution for the Problem 3-2.

In the next chapter, we shall discuss some variations in the meaning altering transformations.

```
{I}
 k,i, head2, head3 := 0,1,1, N + 1;
{P}
 while k ≠ N do
 begin
    if A[i] = 1 then A[i], A head2  := A[head2], A[i];
                     head2 := head2 + 1; i := i + 1;
    else if A[i] = 2 then i := i + 1;
    else if A[i] = 3 then head3 := head3 - 1;
                          A[i], A[head3] := A[head3], A[i];
    fi
    k := k + 1;
 end
{P ∧ (k = N)}
{R}
```

Figure 3-11: A solution for the Problem 3-2.

```
{I}
i, head2, head3 := 0,1,1, N + 1;
{P^3}
while i ≠ head 3 do
begin
   if A[i ]= 1 then A[i] , A[head2] := A[head2], A[i] ;
                    head2 := head2 + 1; i := i + 1;
   else if A[i ]= 2 then i := i + 1;
   else if A[i] = 3 then head3 := head3 - 1;
                         A[i], A[head3] := A[head3], A[i];
   fi
   k := k + 1;
end
{P^3 ∧ (i = head3)}
{R}
```

Figure 3-12: Final solution for the Problem 3-2.

4. VARIATIONS

In the previous two examples, we were able to combine the solutions for the simpler problems without much difficulty because the solutions meshed together nicely. This may not always be the case and some other strategy must be found to combine the solutions to simpler problems. The following example, illustrates two strategies for combining the programs This example has been adopted from Lucena.[10]

Example 4.1 PRINTING FROM CARDS

Text punched on computer cards is to be printed on a line printer. All cards except the last one contain 80 characters. The last card has characters upto and including the character '@', indicating the end of the text. Each line printed must contain 132 characters.

One may naturally divide the above problem into two subproblems : one for reading a card and the other for assembling and printing a line If we code this two subproblems, Figures 4-1 and 4-2 come to mind.

We observe that the two subproblems are simply implemented through counting loops. However, it is not straight forward to combine the two solutions, since the input and output records have different lengths. We may retain one of the solutions and unfold the loops of the other, while combining them. For example if the solution for the 'assembling and printing' is retained, we get the program in Figure 4-3.

There is another way of combing these solutions. The two processe for input and output can run in parallel, provided the k^{th} character is assembled in the output record before the $k+1^{st}$ character is selected

```
integer ; character array card [1:80];
character char;
repeat
   read (card); i := 1;
   repeat
      char := card[i] ;
      process char
      i := i + 1;
   until (i > 80) V (char = '@')
until char = '@';
```

Figure 4-1: Read a Card

```
integer j; character array line [1:132];
character char;
repeat
   j := 1;
   repeat
     {obtain one character}
      line[j] := char;
      j := j+1;
   until (j > 132) V (char = '@')
   while j ≤ 132 do line[j] := ' '; j := j + 1;
   print (line);
until char = '@';
```

Figure 4-2: Assemble and print a line.

```
integer i,j;
character array card [1:80] ; line [1:132] ;
character char;
read (card); i := 1;
repeat
   j := 1;
   repeat
      if i > 80 then read (card); i := 1; fi
      char := card [i ];
      line [ j ] := char;
      i := i+1; j := j+1;
   until (j > 132) V (char ='@')
   while j ≤ 132 do line [j ] := ' '; j := j+1;
   print (line);
until char ='@'
```

Figure 4-3: A complete solution for printing from cards.

```
buffer . monitor
   begin
      character bufchar;
      boolean full;
      condition nonempty, nonfull;
      procedure givechar (x : character);
      begin
         if full then nonfull. wait. fi
         bufchar := x;
         full := ¬full;
         nonempty. signal;
      end givechar;
      procedure getchar (x : character);
         begin
            if full then nonempty. wait; fi
            x := bufchar;
            full := ¬full;
            nonfull. signal;
         end getchar;
      full := false;
   end buffer
```

Figure 4-4 (Part I): A parallel solution for printing from cards.

```
cobegin
   input  : begin
               integer i;
               character array card [ 1 : 80 ];
               character inchar;
               repeat
                  read (card); i := 1;
                  repeat
                     inchar := card [ i];
                     buffer. givechar  (inchar);
                     i := i+1;
                  until (i > 80) V (inchar = '@')
               until inchar = '@'
            end
output : begin
            integer j;
            character array line [1 :132 ];
            character outchar;
            repeat
               buffer. getchar (outchar); line [j]  := outchar;
               j := j+1;
            until (j > 132) V (outchar = '@')
            while j ≤ 132 do line [j ] := ' '; j := j+1;
            print (line);
            until outchar = '@'
         end
coend
```

Part II

Figure 4-4: A parallel solution for printing from cards.

abstraction at high level, one can easily express this timing inter-relation between the two processes. A solution using monitors (Hoare,4), to express this inter relation, appears in Figure 4-4.

The above example shows that integration of programs may become a difficult task, and for some programs manual integration of programs ma be prone to errors. We may need a programming environment aiding in systematic transformations in order to practically program in this mann

Until now, we have been working on procedure dominated problems. There is a class of problems, where the efficiency of the solution depe ds solely on the data structure selected. It would be natural to explo whether meaning altering transformation could be used to develope attri butes of a data structure, in order to efficiently solve a problem. Th following example illustrates that it is indeed possible.

Example 4-2

Large sets of records are usually organized as binary trees to facilitate efficient searching of bugs. Since different organizations of binary trees give different average performance for searching of a t it is natural to ask about the best possible tree for searching a table of keys with given frequencies. For example, the optimum tree for 31 most common English words is shown in Figure 4-5 (8), it requires only 3.437 comparisons for an average successful search.

The Hu-Tucker algorithm (5,8) constructs such a binary tree for the special case, when a key will always be found. Phase 1 of the algorithm constructs a binary tree, which can then be transformed into an optimum binary tree. If appropriate data structures are used, this

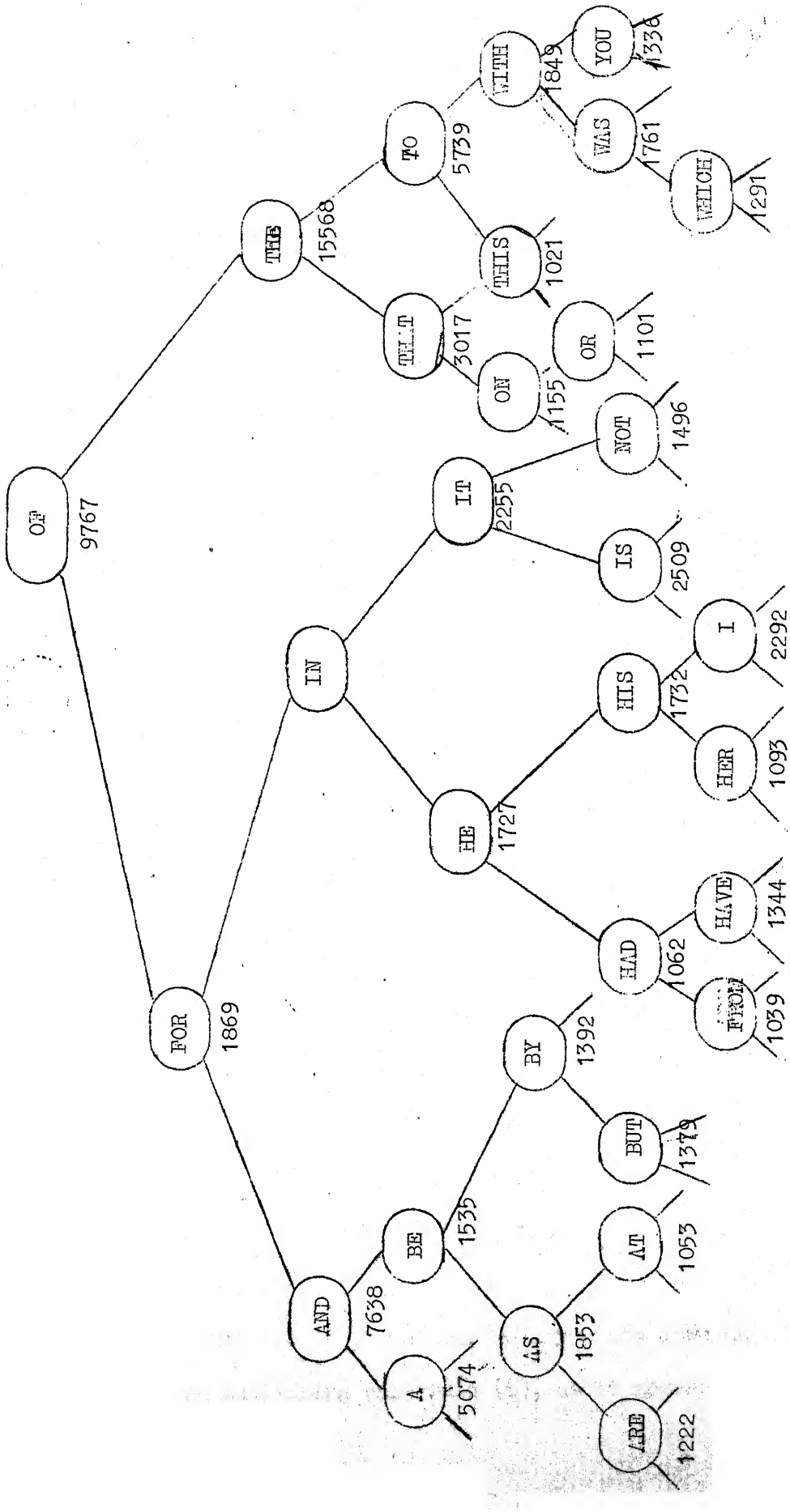

Figure 4-5: Optimum search tree for the 31 most common English words.

The problem is to develop the data structures for the Phase-1 of this algorithm. This phase of the algorithm is given below. In the following, the external nodes represented keys and the respective weights their frequency of occurrence.

Step 1: Start with a working sequence of weights written inside of external nodes, $q_0 \ldots q_n$.

Step 2: Repeatedly combine two weights q_i and q_j, for $i < j$ into a single weight $(q_i + q_j)$ deleting the node containing q_j from the working sequence and replacing the node containing q_i by the internal node containing $(q_i + q_j)$ (4-1)

This combination is to be done on the unique pair of weights (q_i, q_j) satisfying the following rules:

(i) No internal nodes occure between q_i and q_j.

(ii) The sum $(q_i + q_j)$ is minimum over all (q_i, q_j) satisfying rule (i).

(iii) The index 'i' is minimum over all (q_i, q_j) satisfying rules (i) and (ii).

(iv) The index 'j' is minimum over all (q_i, q_j) satisfying rules (i), (ii), and (iii).

The nodes may be identified by a triple (index, q, external). 'index' describes the position of the node in the working sequence and q the weight. If 'external' is true, the node is external, otherwise it is an internal node.

With the above data structure, we may write down an [illegible] of the program satisfying only rule (i), as it appears in Figure 4-6(a).

In the program, a pair of nodes with no internal nodes between them is repeatedly needed. It may be advantageous to keep such nodes in a set. Initially, only two adjacent external nodes will be in such a set.

Once the nodes are arranged in such a structure, they satisfy the following properties:

1. An external node may be a member of at most two sets.
2. An internal node is a member of only one set.
3. When two nodes are combined, node belonging to the sets to which either or both the nodes belong, also have no external nodes among them. Hence, these sets may be replaced by their union.
4. There may be at most 3 such sets.

The program with the refinement in the data structure appears in Figure 4-6(b).

We may, now, apply Rule (ii) of the algorithm to refine the data structure still further. Sum of weights of the selected pair of nodes must be minimum over all such pairs. It follows that the sum must also be minimum over all pairs of nodes belonging to a set. Transforming these sets into ordered sets, ordered in ascending order of node weights will help in selecting a minimum weight sum pair. In addition, these ordered sets may be arranged, in ascending order of respective minimum weight sums, to make the selection of a minimum weight-sum pair trivial.

At this stage, part of the program in Figure 4-6(b) may be transformed to give Figure 4-6(c).

1: 'initialise'

repeat

2: 'Find two nodes with index i and j such that there is no external node between them and i j'

3: 'Remove node with index j, and replace node with index i, with an internal node (i, q_i, q_j, false)'.

until 'one node is left'

(a) Top level abstract program, satisfying only rule (i)

1: 'Prepare sets of nodes with no external nodes between them'

repeat

2.1: 'Select a set and two nodes belonging to it'

2.2: 'Replace the sets to which either or both node belong, with their union'

3 : 'Remove node with index j and replace node with index i, with an internal node (i, $q_i + q_j$, false)

until 'one node is left'

(b) Program with refined data structure, satisfying rule (i), (ii)

2.1: 'Select the first ordered set from the master order set and select the first two nodes from the tuple'

2.2.1 'Replace the ordered set to which either or both of selected nodes belong, with their ordered union'

2.2.2 'Update the master ordered set'

(c) Further refinement to satisfy rules (ii), (iii), and (iv

Figure 4-6: Phase-1 of Hu-Tucker Algorithm.

Applying rule (iii), where minimum index 'i' is required, nodes with equal weights in an ordered set may be arranged in order of their index. Similarly ordered sets with equal minimum weight sums may be arranged in order of index of the first node belonging to them. Rule (iv) may be applied to take care of ties in the above arrangement. The ordered sets with equal weight sums and same first node index, may be arranged in order of the index of the second node beloning to them.

With the above data structure and its attributes, we only need to select its implementation, in order to get a program suitable for compiling.

We have ordered sets and an ordered master set on which following operations are carried out:

1. Add or delete a node (ordered set) in an ordered set (master ordered set).
2. Merge at most 3 ordered sets at every stage.

To efficiently carry out the first operation of addition and deletion an ordered sets (master set) the natural data structure is a binary tree. It takes O(log n) time to add or delete a node from a binary tree. While there are several ways in which a binary tree could be organized, only with a heap or a leftist tree, can one merge two trees in O(log n) time.[8] Thus, the ordered sets and the master ordered set may be implemented as either heaps or lefitst trees to efficiently program the Phase-1 of the Hu-Tucker algorithm.

It is only a mechanical job to transform our program, using standard primitives and implementation of addition, deletion and merging of heaps or leftist trees.[7,8]

Let us summarize the process, adopted in solving the above problem. We simplified the algorithm to aid in finding a suitable arrangement of the information. Then, we gradually added rules one by one, at the same time adding new attributes to the information structure. Finally, having known the attributes of the information structure and the operations on it, we picked an implementation of the information structure, to efficiently carryout the given operations.

The following example illustrates use of meaning altering transformations in hardware implementation of multiplication algorithms. This example has been adapted from Kartashev.(6)

Example 4-3: Multiplication Algorithms

Hardware multiplication circuits are usually implemented as a series of additions and shift operations.

Let R_1 and R_2 be the multiplicand and multiplier registers of length n bits. The product of these two registers will be of length, 2n, and may be stored in R_3, the register in which partial sums are stored. Let Σ be the adder circuit and $\longrightarrow$ be the shift circuit, where the direction of the arrow indicates the direction of the shift.

The following example explains how human beings multiply two numbers.

	1010	R_1
	X 1001	R_2
	00000000	R_3
+1*	1010	$\longleftarrow R_1$
+1*	10100	$\longleftarrow R_1$
+0*	101000	$\longleftarrow R_1$
+1*	1010000	$\longleftarrow R_1$
	01011010	$R_3 = \Sigma$

In order to implement such an algorithm, we need registers R_1 and R_3 of size 2n. This algorithm is described below:

Algorithm A

A_1 : [Initialise] i := 1, R_3 := 0

A_2 : If i = n terminate
Shift right R_2. If least significant bit (LSB) of R_2 is 0, go to A_3, else go to A_4.

(4-2)

A_3 : [Shift R_1 without adding]
Shift left R_1 and go to A_5.

A_4 : [add R_1 to R_3 and shift]
R_3 := R_3 + R_1, shift left R_1.

A_5 : i := i+1, go to A_2.

The hardware realization of the above algorithm appears in Figure 4-7.

If the partial sum register R_3 is shifted right instead of R_1 being shifted left, the effect of algorithm remains unchanged. R_1 may now be added to the most significant bits of R_3, and it is possible that there is an overflow. To take care of this possibility, overflow bit of the adder may be fed to the most significant bit (MSB) of R_3. The advantage of the above transformation is that R_1 need only be of length of n.

The following example illustrates algorithm after the above transformations.

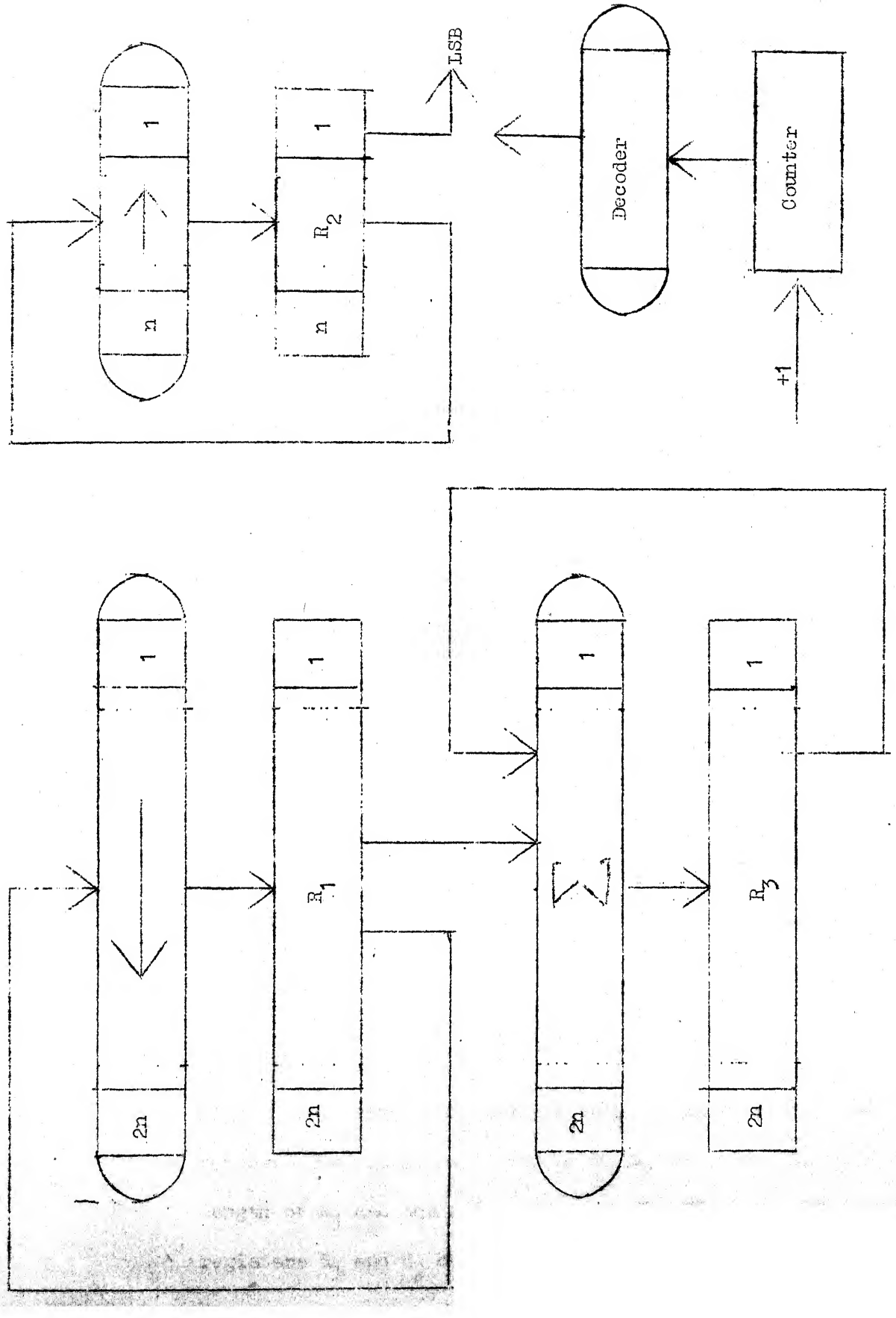

Figure 4-7: Hardware realization of Algorithm A.

	1010	R_1	
	X 1001	R_2	
	00000000	R_3	
+1*	1010	R_1	
	10100000	R_3	= Σ_1
	01010000	R_3	↓
+0*	1010	R_1	
	01010000	R_3	
	00101000	R_3	→
+0*	1010	R_1	
	00101000	R_3	
	00010100	R_3	↓
+1*	1010	R_1	
	10110100	R_3	= Σ_2
	01011010	R_3	↓

The above algorithm has been described below:

Algorithm B

B_1 : [Initialise] i := 1, R_3 := 0

B_2 : If i = n terminate
Shift right R_2. If LSB of R_2 is 0, go to B_3
else go to B_4.

B_3 : [Shift R_3 without adding]
Shift right R_3 and go to B_5. (4-3)

B_4 : [add and shift R_3] $R_3 := R_3 + R_2 \times 2^n$
Shift right R_3, feed the overflow bit of the adder
to MSB of R_3.

B_5 : i := i + 1, go to B_2.

The hardware realization of the above algorithm appears in Fig. 4-8.

In the implementation of above algorithm, at i^{th} stage, left most i bits of R_2 are all zeroes. Also, during addition of R_1 to R_3, right most n bits of R_3 are not modified and rightmost n-i bits are zeroes. The remaining i bits of R_3 may be stored in R_2 to reduce the length of R_3 and the adder circuit. The final result appears in the registers R_3 and R_2 combined.

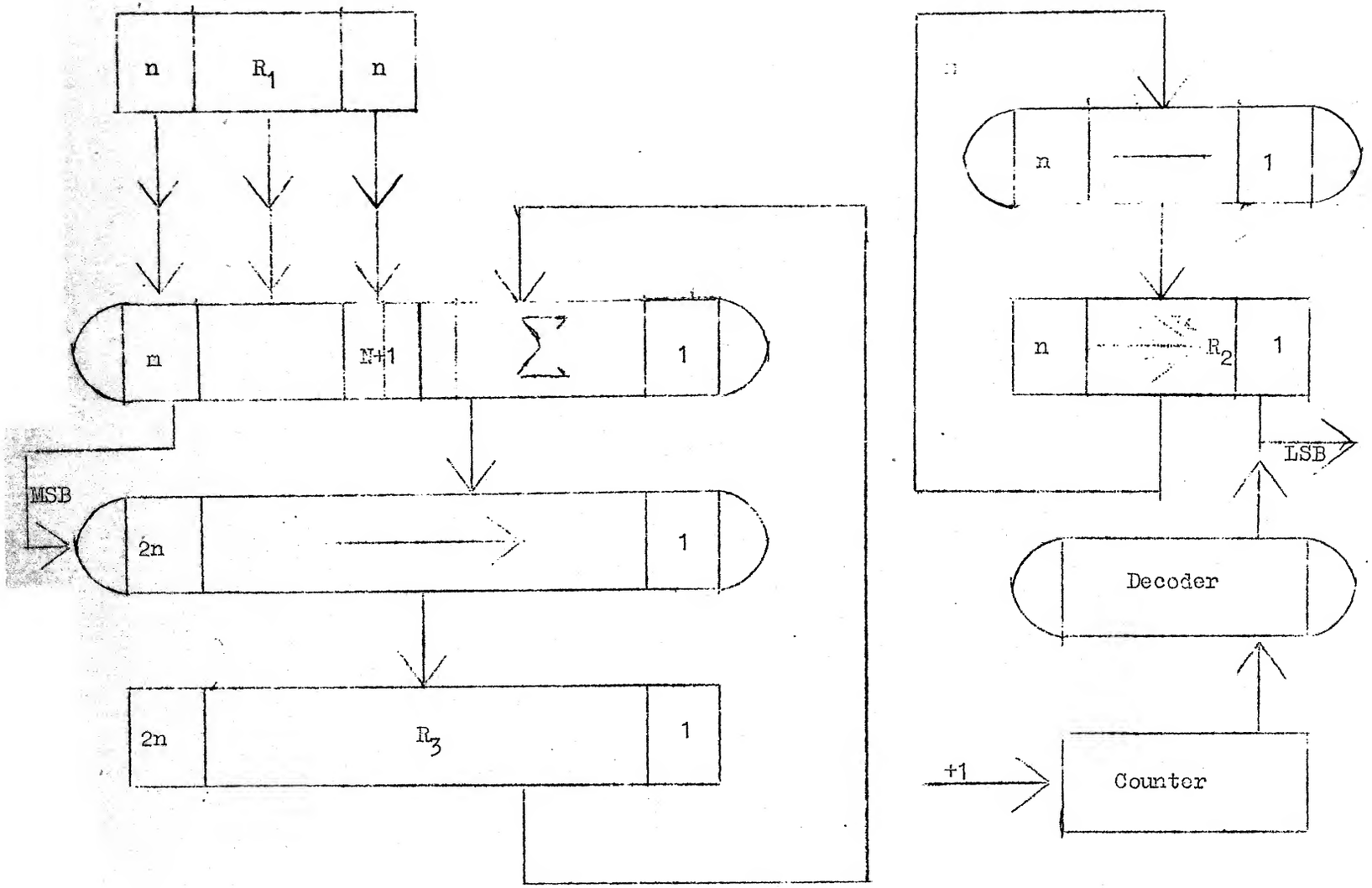

Figure 4-8: Hardware realization of Algorithm B

After the above transformations, the algorithm B appears as algorithm C, below:

Algorithm C

C_1 : [Initialize] i := 1, R_3 := 0

C_2 : If i = n terminate
Shift right R_2. If LSB of R_2 is zero, go to C_3
else go to C_4. (4-4)

C_3 : [Shift R_3 without adding]
Shift right R_3 MSB of R_2 := LSB of R_3
go to C_5.

C_4 : [add and shift R_3]
shift right R_3. Feed overflow bit of the adder and LSB of R_3 to MSB's of R_3 and R_1 respectively.

C_5 : i := i+1, go to C_2.

The hardware realization of the above algorithm appears in Figure 4-9.

In the above example, we successively applied meaning altering transformations to various registers and improved algorithms to reduce the cost of hardware.

The variations of meaning altering transformations described in this chapter, by no means, exhaust the class of such transformations. This class of transformations will have to be augmented in order to be effectively and conveniently used.

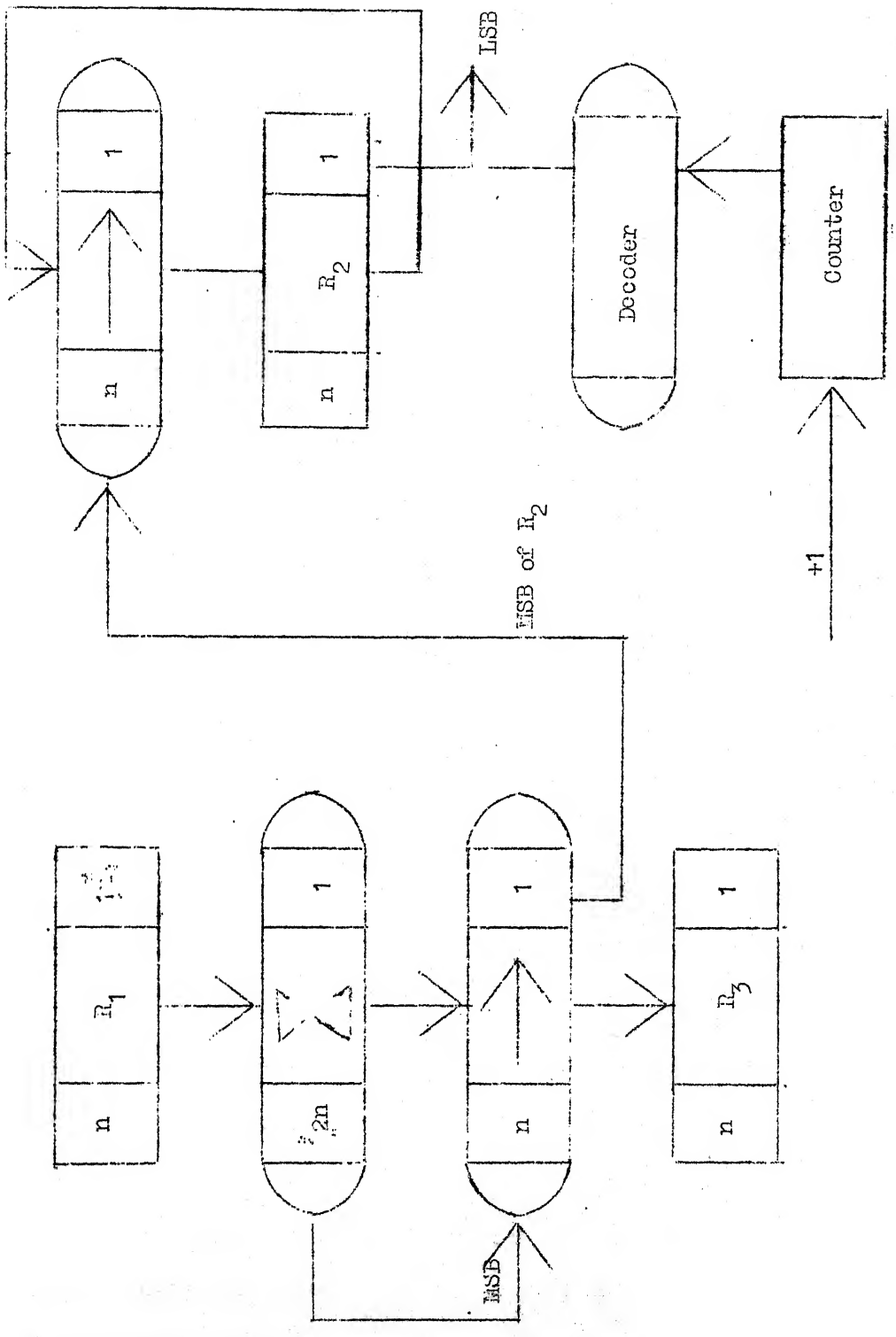

Figure 4.9: Hardware realization of Algorithm C.

5. CONCLUSIONS

Through various examples, we have seen how meaning altering transformations, together with other programming techniques, may be used for the development of a complex algorithm and/or data structure. If a problem is simple, it is easier to program, prove theorems about the program, and understand.it. Easily provable and understandable programs for the related problems are combined or extended to build up the final solution. In parallel with the above process, proofs of correctness of programs being combined, may also be combined to give a proof of correctness of the final solution.

The practicality of structured programming in delivering robust, usable programs is not beyond doubt. As Knuth[9] quotes, "people can argue that structured programs, even if they work correctly, will look like laboratory prototypes where you can discern all the individual components, but which are not daily usable. Building 'integrated products' is an engineering principle as valuable as structuring the design process." He also quotes the plans for a prototype system that will automatically assemble integrated programs from well-structured ones that have been written top-down by stepwise refinement. Such a system, with suitable modifications, would be very useful for programming with meaning altering transformations.

Programming through transformations and integrating programs are complex tasks and may be prone to errors if performed mannually. It would be worthwhile to have such a technique mechanized. Knuth[9] has given the following comments on interactive program manipulation system:

"Programmer using such a system will write his beautifully structured but possibly inefficient program; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one.... The original problem should be retained along with the transformations specifications, so that it can be properly understood and maintained as time passes."

Standish and others[16] have designed a system based on the above comments and the Irvin program transformation catalogue.[15] Such a program manipulation system may be extended by including meaning altering transformations. Such a system, alongwith a system that can combine programs as well as their proofs of correctness, may emerge as a future system to aid in program development, proving and maintenance.

Future Work

1. Catalogue of Meaning Altering Transformations

A few meaning altering transformations have been listed here. They necessarily donot exhaust all possibilities. A catalogue of such transformations would be useful in further strengthening our programming ability.

2. Program Manipulation Systems

A program manipulation system which can change the meaning of programs, needs further research, before it can be implemented. Program integration has to be an essential feature of the system. Various methods of programs storage and program integration algorithms need to be investigated. If programs are to be combined or merged, their file structure need to be different from a linear one. Since some tree structures, e.g., heaps, leftist trees, can be merged very efficiently,[7] it may be advantageous to store programs as one of these structures.

3. Manipulation of Proofs of Correctness of Program

Along with manipulating programs, their proofs of correctness could also be manipulated. When a complex problem is decomposed into several related problems, the input-output assertions of the original problems could also be decomposed to obtain input-output assertions for the related problems. When the related problems are combined or their meaning is changed, same process may be carried out on this proofs. A system which manipulates proofs in this manner may be mechanized. Work in this direction may result in interesting correctness proving techniques.

4. Area of Automatic Program Synthesis

Another related area of investigation may be the area of automatic program synthesis. The automatic program synthesizer of Manna and Waldinger[11] may be extended by a system which simplifies input-output assertions, synthesizes programs for these problems and then combine them to give the final program.

5. Application in Chief Programmer Team Management

The chief programmer team management technique of Baker[1] may be extended by the use of meaning altering transformations. The chief programmer may simplify the given task recursively to yield a set of related programs; then he may give a set of transformations to be applied, to his programmer team. He may explain the use of such a transformations on trivial example problems.

LIST OF REFERENCES

1. Baker, F.T., "Chief programmer team management of production programming", IBM Syst. J., 11, 1(1972), 56-73.
2. Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R., "Structured Programming", Academic Press, London, England, 1972, pp. 220.
3. Hoare, C.A.R., "An axiomatic approach to computer programming", Comm. ACM, 12, 10 (October 1969), 576-580, 583.
4. Hoare, C.A.R., "Monitors : an operating system structuring concept", Comm. ACM, 17, 10 (October 1974), 549-557.
5. Hu, T.C., and Tucker, A.C., "Optimal computer search trees", SIAM J. Applied Math., 21, 1971, 514-532.
6. Kartashev, S., "Computer System Organization : Synopsis", Department of Computer Science, University of Nebraska, Nebraska, 1976.
7. Knuth, D.E., "Fundamental Algorithms : The Art of Computer Programming" Vol. 1, Addison-Wesley, Reading, Mass., 1968, 2nd Edition, 1973, pp. 634.
8. Knuth, D.E., "Sorting and Searching : The Art of Computer Programming", Vol. 3, Addison-Wesley, Reading, Mass., 1973, pp. 722.
9. Knuth, D.E., "Structured Programming with go to Statements", Computing Surveys, 6, 4 (December 1974), pp. 261-301.
10. Lucena-Filho, G.J.,"The role of temporal abstraction in programming : A Thesis Proposal, University of Waterloo, Waterloo, Ontario, June 1976, pp. 35.

11. Manna, Zohar, and Walinger, Richard J., "Towards automatic program synthesis" in Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, 188, E. Engeler (Ed.), Springer-Verlag, New York, 1971, 270-310.

12. Sahasrabuddhe, H.V., "On Programming Styles", Technical Report No. 12-C-051175, Systems Design Department, University of Waterloo, Waterloo, Ontario, November 1975, pp. 18.

13. Schaefer, Marvin, "A mathematical theory of global program optimization," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973, pp. 198.

14. Spier, Michael, J., "The jigsaw puzzle analogy", Guest Editorial, Software Practice and Experience, 6, 4(Oct-Dec. 1976), pp. 443-446.

15. Standish, T.A., et. al., "The Irvine Program Transformation Catalogue," Department of Information and Computer Science, University of California at Irvine, Irvine, California, 1976, pp. 82.

16. Standish, T.A., "Improving and refining programs by program manipulation", Proceedings National Computer Conference, 1976, pp. 509-516.

17. Wirth, N., "Program development by stepwise refinement," Comm. ACM, 14, 4 (April 1971), pp. 221-227.

18. Wirth, N., "Systematic Programming : An Introduction," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973, pp. 167.

19. Wirth, N., "On the composition of well-structured programs", Computing Surveys, 6, 4 (December 1974), 247-259.